263-2300-00: How To Write Fast Numerical Code

Assignment 2: 100 points
Due Date: Thu March 15 17:00
http://www.inf.ethz.ch/personal/markusp/teaching/263-2300-ETH-spring12/course.html
Questions: fastcode@lists.inf.ethz.ch

Exercises:

1. *Short project info (10 pts)* Submit the following about your project (be brief):

   (a) An exact (as much as possible) problem specification for your class project.
   For example for MMM, one can be very precise, and it could be like this:
   Our goal is to implement matrix-matrix multiplication specified as follows:
   Input: Two real matrices A, B of compatible size, $A \in \mathbb{R}^{n\times k}$ and $B \in \mathbb{R}^{k\times m}$. We may impose divisibility conditions on n, k, m depending on the actual implementation. *Output:* The matrix product $C = AB \in \mathbb{R}^{n\times m}$.

   (b) The algorithm you plan to consider for the problem. You can actually sketch the algorithm or just give the name and a precise reference (e.g., a link to a publication plus the page number) that explains it.

   (c) A very short explanation of what kind of code already exists and in which language it is written.

2. *Performance of Scalar Add (30 pts)* Code needed
   This problem builds on the study of ILP using reductions in Lecture 4. Download, extract and inspect the code. We provide C source and header files in the scalaradd folder that benchmark this kernel under Linux (x86 compatibles).

   (a) Using your computer, compile and run the code. the code outputs a file containing different results for varying number of accumulator variables (K) and unrolling factors (L). Summarize the results in a table similar to the one shown in Lecture 4. For which K do you obtain the maximal performance? Compare against the model $K = \lceil \frac{latency}{cycles\_per\_issue} \rceil$.

   (b) The file funcs.h contains different functions for every possible pair of K and L. Pick the one associated to the best values of K and L for the exercise explained now.

      i. For all vectors of sizes $n = 100 \cdot ceil(1.8^i)$, with $i \in \{1, \dots, 21\}$, create a semi-log plot where the x-axis is in logarithmic scale, and the y-axis in normal scale. The x-axis shows n and the y-axis performance in MFlop/s or Gflop/s.

      ii. You should see that the line consists of piecewise plateaus. Try to explain the plateaus, meaning try to explain the height of the plateaus and where the plateaus transition using suitable hardware parameters. (Hint: remember problem 4 in HW 1.)

Solution: Increasing the number of accumulators we reduce the effect of sequential dependencies in the execution flow. This allows performance to get closer and closer to its limit, which is the computational throughput of the execution unit. We provide answers based on results obtained running the code on an Intel Core i7-2600 CPU @ 3.40GHz.
General remark: when running performance tests it is good practice to turn off possible sources of indeterminism in the measurement, such as power optimization, simultaneous multithreading, frequency scaling and turbo mode functionalities. One can typically set such options in the computer's BIOS.

(a) Performance in Tab. 1 is reported in cycles/flop. Choosing $K = 3$ is enough to fill in the execution pipeline with independent instructions, as described by the model presented in class.

Table 1: Performance of scalar add (cycles/flop) varying the unrolling factor (L) and the number of accumulators (K).

K/L	**1**	**2**	**3**	**4**	**5**	**6**
1	2.977	2.956	2.958	2.955	2.959	2.958
2	0	1.495	0	1.495	0	1.495
3	0	0	1.016	0	0	1.018
4	0	0	0	0.016	0	0
5	0	0	0	0	0.018	0
6	0	0	0	0	0	1.018

(b) The height and length of the plateaus in Fig. 1 relates to the different levels of memory and their bandwidth. In our specific case, the three levels of cache are able to provide data at a rate that allows the execution unit to compute very close to peak performance (1 flop/cycle · 3.4 GHz = 3.4 GFlop/s). On the other hand, the LLC-DRAM link has a bandwidth of 0.5 double/cycle. As a consequence, we are bound to half of the peak performance (1.7 GFlop/s) when fetching data from DRAM. However, we note a 10% drop in performance for sizes larger than the L2 cache. This is due to other factors, such as the overhead of translating virtual addresses. Some of this issues we will address later on during the course.

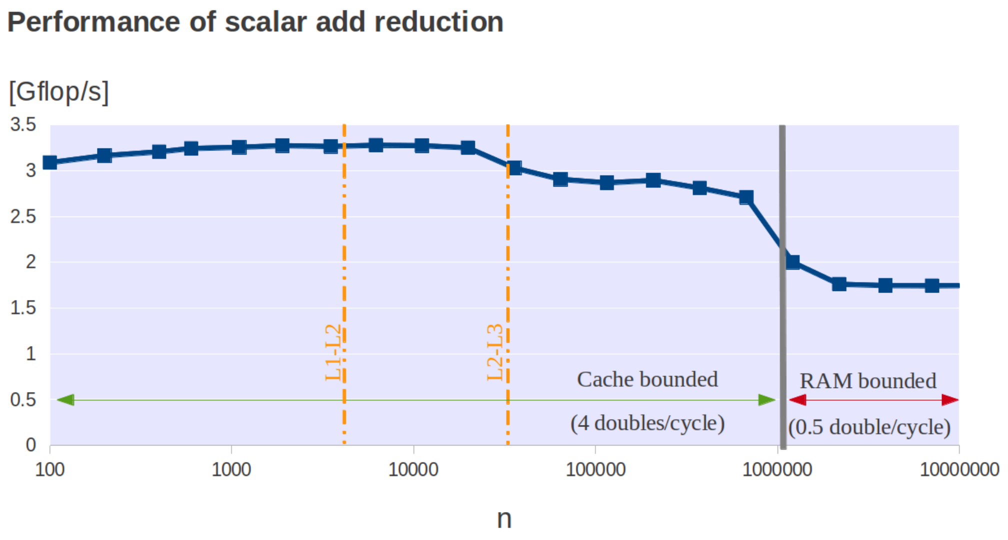

Figure 1: Performance of scalar add reduction with working set $n = 100 \cdot ceil(1.8^i)$, $i \in \{1, \ldots, 21\}$.

3. *Optimization Blockers (30 pts)* Code needed
   Download, extract and inspect the code. Your task is to optimize the function called superslow (guess why it's called like this?) in the file **comp.c**. The function runs over an $n \times n$ matrix and performs some computation on each element. In its current implementation, *superslow* involves several optimization blockers. Your task is to optimize the code.

   Edit the Makefile if needed (architecture flags specifying your processor). Running `make` and then the generated executable verifies the code and outputs the performance (for this the op count is underestimated by $2n^2$) of superslow. Proceed as follows

   (a) Identify optimization blockers discussed in the lecture and remove them

   (b) For every optimization you perform, create a new function in `comp.c` that has the same signature and register it to the timing framework through the *register_function* procedure in *comp.c* Let it run and, if it verifies, determine the performance.

(c) In the end, the innermost loop should be free of any procedure calls and operations other than adds and mults.

(d) When done, rerun all code versions also with optimization flags turned off ($-O0$ in the Makefile).

(e) Create a table with the performance numbers. Two rows (optimization flags, no optimization flags) and as many columns as versions of superslow. Briefly discuss the table.

(f) Submit your `comp.c` to the SVN

What speedup do you achieve?

Solution:

Figure 2: Performance of the submitted code

4. *Locality MVM(10 pts)*

Consider the following straightforward code for Matrix Vector Multiplication (MVM) computing $y = Mx$, where both x and y are of length n.

```
void MVM(double *M, double *y, double *x, int n)
{
  for (int i = 0; i < n; i++) {
    y[i] = 0;
    for (int j = 0; j < n; j++)
      y[i] += M[i*n + j] * x[j];
  }
}
```

Inspecting the array accesses, where do you see

(a) Temporal locality?

(b) Spatial locality?

Solution: $y[i]$ clearly shows temporal locality as it is accessed every j iteration n times. Many people forgot about $x[j]$, which given a big enough cache, will also have temporal locality. M and $x[j]$ clearly have spatial locality as they are walked sequentialy in the inner loop, but also y is accessed in a sequential manner. The code itself shows both localities.

5. *Operational Intensity of a Program (15 pts)*

Given the following function

```
void Corr(double *x, double *y, double *h, unsigned int k, unsigned int n)
{
  int i, j;

  for(i = 0; i < n - k + 1; ++i ) {
    y[i] = 0.0;
    for(j = 0; j < k; ++j )
      y[i] += x[i + j] * h[j];
    }
}
```

x and h are the input arrays of length n and k, respectively. y is the resulting array which has length $n - k + 1$.

(a) Determine the cost of this algorithm (cost measure = number of floating point ops).

(b) Determine an upper bound for the operational intensity.

Solution:

$$I = \frac{\text{Number of operations}}{\text{Data transferred LLC} \leftrightarrow \text{RAM}} = \frac{2k(n-k+1)}{\underbrace{n+k}_{input} + \underbrace{n-k+1}_{output}} = \frac{2k(n-k+1)}{2n+1} \in \mathcal{O}(k) \qquad (1)$$